Increasing Software Availability and Scalability with Microservices Architecture

Mattias Nixell

Bachelor's Thesis in Information Technology
Vaasa 2019

THESIS

Author:	Mattias Nixell
Degree Programme:	Information Technology, Vaasa
Supervisors:	Elisa Nyström, Stockmann Oyj Abp
	Kaj Wikman, Novia University of Applied Sciences

Title: Increasing Software Availability and Scalability with Microservices Architecture

Date May 6, 2019 Number of Pages 23

Abstract

Stockmann's e-commerce system's maintainability and scalability issues have become more apparent inside the company since their recent announcement of a "digital acceleration". A microservice architecture was implemented in one of Stockmann's e-commerce sub-systems as a potential architectural approach to solve these issues. It was chosen due to its promoted architectural style of breaking down a system into smaller and more independent pieces, believed to produce a system that is more maintainable and scalable. The microservice architecture was implemented with software design concepts, strategies and principles advocated in related literature. The architectural choice enabled the sub-system to perform at significantly higher efficiency with a superior capability of automatic scaling. In addition, the sub-system was perceived to be more maintainable, testable and observable by the developers. The results provided Stockmann with confidence to invest further into microservice architecture and made the development team feel ambitious towards the approach.

EXAMENSARBETE

Författare:	Mattias Nixell
Utbildning och ort:	Informationsteknik, Vasa
Handledare:	Elisa Nyström, Stockmann Oyj Abp
	Kaj Wikman, Yrkeshögskolan Novia

Titel: Öka tillgänglighet och skalbarhet i mjukvara med mikrotjänst-arkitektur

---

Datum 6.5.2019 Sidantal 23

Abstrakt

Stockmanns offentliga meddelande om en ”digital acceleration” har internt uppenbarat underhålls- och skalbarhetsproblem i deras e-handelssystem. I ett av Stockmanns e-handelsundersystem har en mikrotjänst-arkitektur implementerats för att lösa dessa problemen. Arkitekturen valdes eftersom den hade potential att bryta ner ett system i mindre och självständigare delar för att gynna underhåll och skalbarhet. Mikrotjänst-arkitekturen har implementerats med hjälp av koncepter, strategier och principer som främjats i relevant litteratur för planering av mjukvara. Arkitekturen gav undersystemet en överlägsen prestationsförmåga både i effektivitet och skalbarhet. Dessutom upplevde utvecklarna att undersystemet var lättare att underhålla, testa och observera. Resultaten vann Stockmanns förtroende att investera mera i mikrotjänst-arkitektur och utvecklarna känner sig ambitiös inför dess framtid.

OPINNÄYTETYÖ

Tekijä:	Mattias Nixell
Koulutus ja paikkakunta:	Tietotekniikka, Vaasa
Ohjaajat:	Elisa Nyström, Stockmann Oyj Abp
	Kaj Wikman, Ammattikorkeakoulu Novia

Otsikko: Ohjelmiston saatavuuden ja skaalautuvuuden parantaminen mikropalvelu-arkkitehtuurilla

---

Päivämäärä 6.5.2019 Sivumäärä 23

Tiivistelmä

Stockmannin äskettäinen tiedote "digitaalisesta kiihdytyksestä" on tuonut esille ylläpidettävyyteen ja skaalautuvuuteen liittyviä vaikeuksia firman sisällä. Näiden ongelmien ratkaisemiseksi mikropalvelu-arkkitehtuuri toteutettiin yhdessä Stockmannin alajärjestelmistä. Tämä arkkitehtuuri-tyyli valittiin johtuen sen kyvystä hajottaa järjestelmä pienempiin ja itsenäisempiin osiin, jonka on mainostettu pystyvän luomaan paremmin hallittava ja skaalattava järjestelmä. Mikropalvelu-arkkitehtuuri toteutettiin kirjallisuudessa suositelluilla ohjelmisto-suunnittelu konsepteilla, strategioilla ja periaatteilla. Arkkitehtuuri-valinta mahdollisti alajärjestelmän selvästi paremman tehokkuuden ja kyvyn automaattiseen skaalautuvuuteen. Lisäksi kehittäjät kokivat alajärjestelmän olevan paremmin hallittava, testattava ja tarkkailtava. Tulosten ansiosta Stockmann on jatkossakin luottavainen sijoittamaan mikropalvelu-arkkitehtuuriin ja kehitys-tiimi on entistä innokkaampi lähestymistapaa kohtaan.

Table of Contents

1 Introduction

Microservice architecture is a software architectural style that aims to break down software systems into small and independent components, a.k.a services. The idea is to improve a system's modularity to potentially make it easier to understand and maintain. This thesis aims to identify the benefits and drawbacks of adopting this architectural style in Stockmann's e-commerce system. In addition, microservice's history and design recommendations are explored to gain insight into its architecture. Stockmann has invested in upgrading a product data import for the e-commerce application, stockmann.com, in being able to prove the potential benefits of moving towards microservices.

1.1 Employer

Digia Oyj is an IT services company with roots from SysOpen Oyj founded in 1990, and Digia Oy founded in 1997. The companies merged into SysOpen Digia Oyj in 2005 and were renamed to Digia Oyj in 2008. Digia advertises itself as a "truly visionary partner in the development of digital business operations and day-to-day processes to data analytics, digital services and e-commerce solutions." (Digia Plc, 2018b). Digia's strategy is to develop and innovate solutions together with its customers in long-term relationships. It also seeks to strengthen itself in IT market areas that allow it to grow faster than the traditional IT market (Digia Plc, 2018a). The acquisition of Igence in 2016, an e-commerce software company founded in 1997, makes Digia a reliable partner for Stockmann's digital retail.

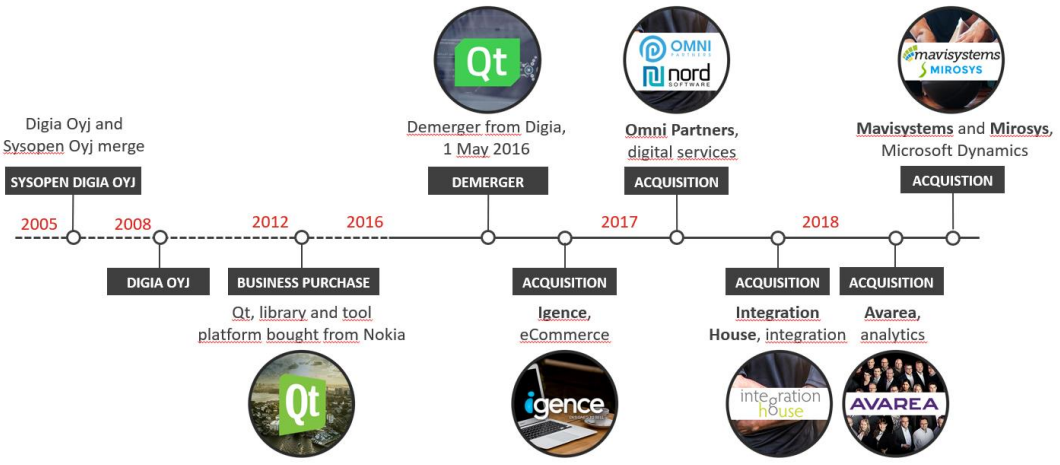

Figure 1. History of Digia Oyj shows their recent aim for expansions into new market areas. (Digia Plc, 2018b)

1.2 Customer

Stockmann is a retail company established in 1862 and known for its high-quality products and service. They aim to be the "NO 1 source of inspiration for modern, urban life" with focus on the customers; "Stockmann is the customers". Stockmann is divided into three divisions; Lindex, Stockmann Retail and Real Estate. Overall, Stockmann's selection is focused on fashion, beauty and home, where fashion is by far the largest product area. (Stockmann plc, 2018, pp.7–11)

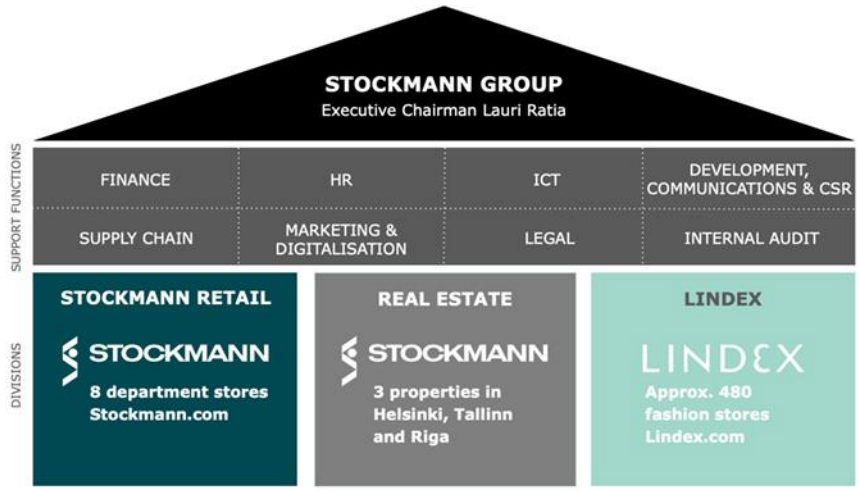

Figure 2 Stockmann's company structure as of 2019. (Stockmann plc, 2019)

Digitalization is an important trend in Stockmann's retail, as the CEO Veijalainen (2018, p. 2) announced; "In Retail, Stockmann also launched a digital acceleration project at the beginning of 2018, with the aim of increasing e-commerce and reinforcing the omnichannel approach.". Stockmann Retail has partnered with Digia Oyj to continue the development of the e-commerce application, stockmann.com, as an enabler of the digital acceleration. The application, initially developed by a German company AOE, is built with the e-commerce platform Magento and powered by AOE's search engine product Searchperience (AOE GmbH, 2017, p.1). It is tightly integrated with a central data communicator, which keeps Stockmann's systems in sync and make a seamless customer experience possible.

1.3 Product Integration

This thesis focuses on a product data integration between the central data communicator and the e-commerce system. The integration is primarily a continuous flow of product data imported

into stockmann.com. The central data communicator continuously sends notifications about product changes to the e-commerce system. In the e-commerce system, the product change notifications enter a queue, which acts as temporary storage until the product integration is available to process the change. The wait time should not be longer than a minute in a healthy environment. The product integration is composed of four applications, which are tightly chained together as seen in the flowchart (Figure 3). In the flowchart, the rounded rectangles are starting points, the laying cylinders are temporary storages, the sitting cylinders are databases, and the blue rectangles are processes (in this case applications).

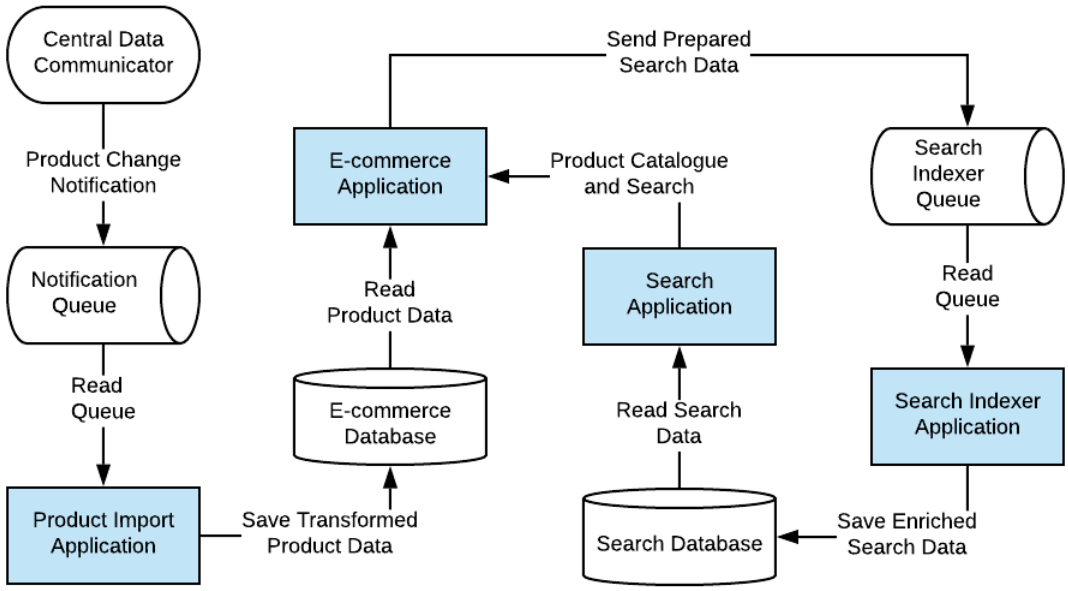

Figure 3 Product integration flowchart to display the relations between the four applications.

The **product import application** processes the notifications. A received notification contains an identification of a changed product, which can be used to fetch all related data from the central data communicator, e.g. product data, categories and promotions. The application's role is to transform the related data into a product object and save it into a database shared with the e-commerce application. In addition, the application can run as a full product import by feeding it a generated *nightly full dump* of data.

The **e-commerce application** (stockmann.com) contains a periodically executed background process, which prepares recently changed products for search capabilities. It maps each product onto a new format based on configurations set in the e-commerce application, which is sent to

another queue to be further processed by the **search indexer application**. The role of the indexer is to make searchable documents out of received data. It works in stages to extract searchable terms and boost product's search relevance based on rules defined in a search admin panel. When the searchable document is ready, it is saved into a search database. Finally, the **search application** provides a set of endpoints which connects the search database with the e-commerce application and enable browsing and searching of the product catalogue.

1.4 Problem Space

For a product to become successfully imported, it must sequentially move through each application, each comprised of a large codebase and its logic contributing to the format of the product data. The current chained composition makes the data logic very entangled, where a feature change introduced at the beginning of the chain often requires a change in the proceeding applications as well. The unfortunate state of the product integration is that it is difficult to understand and slow to develop further.

The announced digital acceleration is expected to increase traffic to the e-commerce website and introduce changes to the product data structure. This placed the team in an awkward position because keeping up the product integration pace would be challenging. The team had also witnessed issues concerning data traffic spikes, where the product integration had become dysfunctional. The product catalog had occasionally shown significantly out-dated product data, e.g. out-of-stock products were shown to the customers. The issues would only become more visible with increased traffic.

It is apparent to involved parties that the product integration is problematic to maintain. Most suggested changes end up stale in the development backlog. Changes are seen to be too costly to implement in relation to the respective business value it would potentially give. The only implemented changes were such that had become imperative to keep the product integration functioning to an acceptable degree. The team must find an approach to make the integration easier to understand and maintain to keep up with the changing business.

2 Literature Review

The team had a vague understanding of microservices before the project. However, it seemed like an appealing approach to solving the problem, because it promotes faster delivery, improved maintainability and raised testability by building small services. In this chapter, the function of the architectural style is introduced by exploring its predecessors and design practices. The scope of the chapter is to understand what defines a microservice and discover means to build upon that definition.

2.1 Brief History of Microservices Architecture

2.1.1 Monolithic Architecture

A monolith, by definition, is “a single block or piece of stone of considerable size, especially when used in architecture or sculpture.” (Dictionary.com, 2018). Like its origin, it is a type of architecture in software development where all the application’s “services” stays as a single package. The architecture is usually straightforward for a fresh developer to get into, but it naturally comes with limitations in development agility and market flexibility.

Dhiman (2015) points out its simplicity and advantage in that “you can run the entire code base on one machine, so when I'm developing and testing, I could probably replicate the entire environment on my machine, because it is just one thing to replicate and configure.”. He also states it is a monolith’s only advantage over a microservices architecture. However, the characteristic is one of the main reasons why a monolithic architecture has been a typical choice in the past with no shortcuts in managing distributed computing.

It is arguably a good idea to take a monolithic approach for a greenfield project. It is common to choose a software framework or a ready-to-use platform, which in most cases is monolithic out-of-the-box. It is also preferable when developing from scratch as the team probably does not have a good sense of the business domain early on. Fowler (2015) started calling this greenfield approach “monolith-first” after his observation of multiple success stories starting with monolithic architecture in contrast to microservices architecture.

2.1.2 Service-Oriented Architecture

As the application grows, the team eventually faces difficulties staying competitive as feature delivery decelerate, deployments become daunting, and hardware scaling is costly. In the early 2000s, the rise of web service standards popularised a service-oriented architecture (SOA) to overcome these obstacles. Erl (2016, pp.46–47) implies that "service-orientation" in SOA takes inspiration from the history of civilization and its service-driven organization. He further defines service as a distinctive task carried out by an individual or collectively by a group of individuals in support of others.

SOA's design follows a similar idea, where a service is a distinct and relatively isolated part of the application. In other words, a service provides a functional context which consists of one or more capabilities. Services in SOA communicate with service contracts, which is a guarantee of functional capabilities a service provides. A service contract is an API published internally for other services of the application, or externally for third-party applications. The services commonly communicate through an enterprise service bus (ESB) in SOA, which acts as a central hub for data transportation.

Erl (2016, pp.58–61) identifies eight fundamental principles which should be carefully adapted to achieve SOA successfully. His principles apply primarily at service-level and he (2016, pp. 441-453) describes them as follows:

- "Services share standardized contracts."
- "Services are loosely coupled."
- "Non-essential service information is abstracted."
- "Services are reusable."
- "Services are autonomous."
- "Services minimize statefulness."
- "Services are discoverable."
- "Services are compostable."

The principles are widely known in enterprise architecture and can be found detailed in a multitude of articles. Unfortunately, design misconceptions and overthinking services led to a failed promise for many companies. Newman (2015, pp.8–9) thinks the pitfalls associated with

SOA originates from the lack of help translating the conventional wisdom into real-world practicalities.

2.1.3 Microservices Architecture

Over the past decade, microservices architecture (MSA) emerged as a variant to the idea of dividing the application into services. Some advocates have referred to it as “a fine-grained SOA”, which makes it natural to compare the two. However, Fowler and Lewis (2014) raise the problem of this comparison by pointing out “SOA means too many different things”. They conclude that “the fact that SOA means such different things means it's valuable to have a term that more crisply defines this architectural style.”. Nonetheless, SOA and MSA both share a similar vision of overcoming monolithic architecture but arguably do so distinctively from one and another.

Newman (2015, p.2) characterizes microservices as small, autonomous services that work together. He gives pointers on the size of microservices to be “small enough, no smaller” and “If the codebase is too big to be managed by a small team, looking to break it down is very sensible.”. He also points out that services in MSA need to be able to change independently of each other. What he means is that one can individually deploy and replace services without forcing other services to change. The benefits of these characteristics are that every service can utilize distinct technology stacks and be replaceable with any other technology stack at any given time. They enable teams to completely rewrite services within a couple of weeks to improve unforeseeable bottlenecks in the system.

However, the smaller one goes, the more complex the big picture gets as one is introducing many moving parts into the system. “The hallmark of a microservice architecture might be smaller services, but following the microservices way will require you to think big. You’ll need to tune your culture, organisation, architecture, interfaces, and services in just the right way to gain the balance of speed and safety at scale.” (Nadareishvili et al., 2016, p.40).

2.2 Microservices Design

Some concepts and principles help you overcome the introduced complexity of moving parts in a microservice architecture. This thesis introduces the ones that were considered most throughout the architecting and development of the integration project. They aim to solve

different problems in software design, but together they become a powerful orchestration that helps the team shape well-coordinated microservices that forms the desired software.

2.2.1 Domain-Driven Design

Eric Evans (2003) introduced the concept of domain-driven design (DDD) as an approach to software design where software is reflected upon the business domain it attempts to enhance. According to him (2003, p.xxi) “the most significant complexity of many applications is not technical. It is in the domain itself, the activity or business of the user.”. DDD aims to align the technical experts with the domain experts by providing a strategic design towards solving complex problems. Microservices design discussions commonly refer to bounded contexts and context maps, which are two central concepts of DDD.

As commonly acknowledged, Evans (2003, p.336) points out that “Multiple models are in play on any large project. Yet when code based on distinct models is combined, software becomes buggy, unreliable, and difficult to understand”. Evans solution to that problem is bounded contexts which explicitly define boundaries of specific parts in software. Vernon (2016, p.32) concisely describes the bounded context as “a semantic contextual boundary. This means that within the boundary each component of the software model has a specific meaning and does specific things. The components inside a Bounded Context are context specific and semantically motivated.”. To accomplish a semantic contextual boundary as a team, Evans proposes to shape a definition of communication he calls ubiquitous language.

Ubiquitous language is an agreed common and accurate way of communicating between participants of a particular context. The idea is to avoid misinterpretation between parties by defining wording that all participants understand the same way. “The software model inside the context boundary reflects a language that is developed by the team working in the Bounded Context and is spoken by every member of the team that creates the software model that functions within that Bounded Context.” (Vernon, 2016, p.34). Pacheco (2018, p.10) mentions that “DDD would be very useful for avoiding misunderstandings in the interpretation of how microservices should work.” after introducing a small software project, suggesting that even a small project benefits from a ubiquitous language.

A bounded context explicitly defines boundaries of one specific part in software. It does not know about other bounded contexts within that software, which means that with the bounded contexts alone, one will not recognize the big picture of the software. The big picture can be easier realized with ”a document which outlines the different Bounded Contexts and the relationships between them” (Avram and Marinescu, 2006, p.73), which is called a context map in DDD. The idea is to find out what kind of relationship there is between bounded contexts, e.g. what kind of translation a model needs to conform to the related bounded contexts.

2.2.2 Loose Coupling and High Cohesion

Ingeno (2018, p.197) believes that “software that is well designed is orthogonal in that its modules are independent of each other. Ideally, changes to one module in a software system should not require changes to another module”. The idea becomes a crucial aspect in MSA where one would consider the modules as independently deployable services. Especially in a frequently changing environment, where teams might be required to introduce multiple changes per day. Ingeno (2018, p.197) considers two measurements for achieving independence; “Orthogonal systems are designed so that their elements are loosely coupled and highly cohesive.”.

Len, Clements and Kazman (2012, p.184) explain coupling as a measurement of “the probability that a modification to one module will propagate to the other”. The services are tightly coupled if the probability is high. The problem with a tightly coupled architecture is that it is challenging to keep track of services that breaks when introducing a change. In the worst-case scenario, the detection of such an effect has already cascaded into a system-wide failure causing significant damage to the business. The goal should be to reduce the number of dependencies between services to achieve a loosely coupled architecture.

“Cohesion measures how strongly the responsibilities of a module are related.” (Len, Clements and Kazman, 2012, p.184). In other words, a service has low cohesion if it is affecting a wide range of business responsibilities. It is also low cohesion if the programmatical logic is the same but span over multiple business responsibilities. The problem with low cohesion is that relatable business responsibilities and relatable programmatical logic are different dimensions. Introducing changes in a system with low cohesion eventually ends up with a messy collection of exceptional logic. While changes in a highly cohesive service would only affect one

responsibility area, thus not exceptional to that area. The goal should be to divide the software by related business responsibilities as distinct domains most likely evolve at different rates.

Finally, these principles align well with the bounded context as Newman (2015, p.33) points out "if our service boundaries align to the bounded contexts in our domain, and our microservices represent those bounded contexts, we are off to an excellent start in ensuring that our microservices are loosely coupled and strongly cohesive.".

2.3 Cloud Computing Service Models

Cloud computing has had tremendous growth over the last decade. However, it is not a new paradigm in information technology. Its roots date back to 1950s with IBM's centralized computer that had multiple remotely connected terminals to it, which was called mainframe computing. This type of network is what cloud computing essentially is, a computing service performed remotely (off-premises). (Neto, 2014)

The main benefits of cloud computing are fast access, cost-efficiency, and self-service requiring little to no prior knowledge of hardware management. As Amazon puts it "You can access as many resources as you need, almost instantly, and only pay for what you use." (Amazon, 2018). Cloud computing is commonly known for three service model types, which are called Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS).

IaaS is the primary offering for cloud IT which provides access to computing resources, storage, and networking. Users have the most control over their resources, without needing to care about the hardware. The cloud vendor maintains the hardware. PaaS is the next step, where the cloud vendor also maintains the underlying infrastructure. The model enables users to focus on their application development and deployment. SaaS is typically a user-interface with minimal administrative access to a software program that runs in "the cloud". (Erl, Puttini and Mahmood, 2013, pp.63–73)

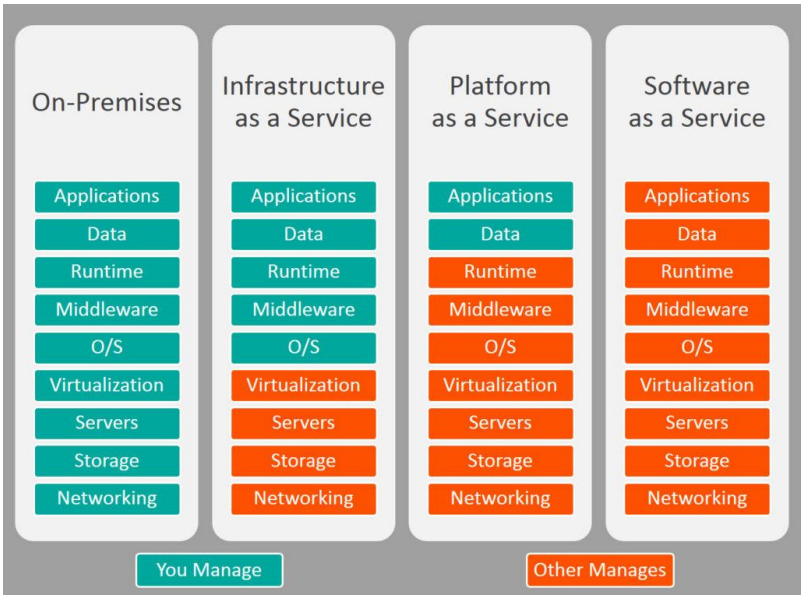

Figure 4. Responsibilities separated by service models. (Watts, 2017)

3 Design and Implementation

The four applications contributing to the product integration each follow a typical monolithic application. They are bundles of features where only fractions of them are needed for each individually imported set of data. To transform into microservice architecture, all those fractions would need to be extracted and organized into a new system while keeping the existing domain logic intact. Stockmann agreed to invest in a design phase to define the product integration domain logic and a preliminary plan for the transformation.

3.1 Bounded Context

As the new product integration would be a significantly different piece of software, a new bounded context was defined that was named "new product flow". A new bounded context made it easier to concentrate its logical boundaries, language and relations to other contexts. Besides, the old product integration spanned of multiple contexts and did not have any real boundaries. A goal was set by the team to isolate all product integration related logic within the bounded context, but maintain the data contracts with the user-facing applications, e-commerce and search.

Previously we found it difficult to communicate problems related to the product integration. A clearly defined language was missing, and it led to miscommunications and time wasted in confusion of communicated problems. We defined a dictionary of terminology as a common language together with Stockmann. In the process, there was a realization that our terminology had been misaligned. The used terms were conflicting or mismatching with other systems, where they could have slightly different meanings. Sometimes, this made it difficult for both parties to form a logical sense in communication. In addition to the dictionary, a table was created to describe the terms' relations to other systems.

With the bounded context, we began exploring the product integration's data logic in the other applications to define which parts would be extracted into the new product flow. We found that it was easiest to start at the end of the flow and find out how each product data field reached its final state. The product data set was enormous. We decided to create a table that describes each field, where the data comes from and how it relates to the final state. The process took several sessions together with Stockmann, but the result was worth it.

With the result, the conclusion was that all the product data logic would be possible to be extracted into the new product flow. This meant that the product import and search indexer application could be removed, and all product integration responsibilities could be moved from the e-commerce application. A significantly shorter path was reached for the product data to reach its final state, as can be seen in the flowchart below. The e-commerce and search application remain, but in comparison to Figure 3 the round-trip to get data into the search application through the search indexer is no longer needed. Instead, the new product flow push transformed product data straight to the search database.

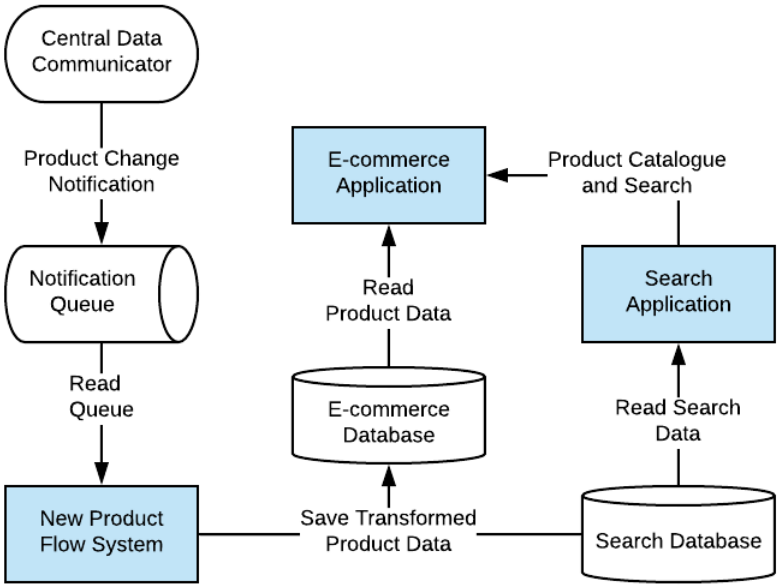

Figure 5 New product integration flowchart with fewer system dependencies.

3.2 Data Storage Isolation

As described in Chapter 1, both the e-commerce and product import applications share a single database. Fowler (2004) describes this as an integration database, where the database integrates data across multiple applications. He emphasizes that many software architects tend to avoid integration database. Newman (2015, p.78) strongly agrees after pointing out (2015, pp.41–42) that strong cohesion and loose coupling is lost with an integration database. Also, Morris (2013) observed that “Most shared databases turn into performance bottlenecks that encourage close-coupling as well as creating a single point of failure.”.

A similar observation had been done in the team; the database integration between the two applications had grown into a performance bottleneck. The single point of failure had also been witnessed more frequently over the past months as the Stockmann's data traffic grows. There have been occasions when the e-commerce application had become completely unavailable for an unacceptable timespan. While investigating the utilization of the database, it was concluded that the database software was not the problem. Instead, there seems to be a combination of poor software architecture decisions that have led to both applications doing an excessive amount of data mutations on the MySQL database.

To isolate the failures related to the data storage within the new bounded context a new database was created, which would only contain product integration related data. None of the product integration data was our master data, meaning that there were no requirements to ensure its persistence. In case of failure, it would be possible to re-import all the related data. With relaxed database requirements, a simple schema-less database was chosen as a cache layer for the latest known state of the product related data. As an example of cache, a prepared category tree was stored so that no transformation related to category data is required on each product change. As a plus, with the new database in isolation, it was possible to process the new product flow at a higher rate without significantly affecting other systems.

3.3 Service Cohesion

The old product integration had a low cohesion in terms of related logic, which made investigating issues related to suspected faulty data extremely daunting and time-consuming. The problem was mostly solved by extracting all product data logic into the new product flow. However, a change notification can contain different types of data, where each data type has a different set of data logic that defines how it is transformed. We defined new microservices as data indexers for each data type, e.g. product indexer and category indexer. The responsibility of an indexer is to read the raw form of the corresponding data from the central data communicator and transform it into a format that is ready to be imported into both the e-commerce and search database. The new indexers made it simple to test what the final data would look like by providing an identification of the data to transform.

A data identifier is needed to run an indexer, which can be extracted from the change notifications. We do not want to send all notifications to all indexers, as that would require the

indexer to check if the notification is relevant to itself. The change notifications are received at a single point, which is the notification queue. We defined a new microservice, named change handler, with the responsibility to coordinate to where notifications are sent. All notifications have the same structure with specific identification and type of data, meaning that the responsibility is relatively simple. As an example, if a category change notification is received, that notification is then forwarded to the category indexer.

This far, the defined microservices can import product data, but there is no logic to remove products from the applications. A product should be removed by a change notification or based on rules for product changes. To remove by notification, we defined a new microservice, named product remover, that has the responsibility to remove a product based on provided identification. Removing a product based on rules was tricky. The rules are based on the final state of the product data format, i.e. the removal cannot be determined without duplicating the product indexer logic. We decided to sacrifice the cohesion of the product indexer by adding the rules at the end of its process. The product indexer sends remove notifications to the product remover microservice based on the rules.

3.4 Service Coupling

The new product flow introduced new moving parts into the system. The microservices are small components that work together towards keeping the e-commerce product catalog up-to-date. However, the services' individuality was not defined, i.e. if we connect them as is, they would be highly coupled with each other. There would not be any sane way to maintain such a structure as a failing microservice would instantly affect all other related services.

Similar to the change notifications enter the product integration through a queue, new queues were introduced in between all related microservices. With queues, a failure in one microservice would less likely cascade that failure to other services as the communication can idle in a queue until such failure is resolved. The queues also enable a microservice to be individually deployed, as it does not any longer matter if the service goes offline for a few minutes. A (mostly) fully decoupled network of microservices was achieved with this approach.

4 Results

The results are based on a relative performance comparison between the old and new product integration. Stockmann has requested that no real numbers are shared with the public regarding their data and integrations. The integration was considered healthy if the product data was processed under a minute, which the team observed to be well met in the new product integration in comparison to the old. The results are relative numbers comparing the old and new product integration without revealing exact underlying numbers. The measurements are recorded from a full product import (FPI) from the time that the process starts, until the time all products are updated in both the e-commerce and search applications.

The old FPI is measured by the average process time from past executions. It can be considered as its extreme as running it during the day usually resulted in failures. To be fair, the new FPI's extremes are also needed for the measurements. The new microservices can scale "infinitely" until the point that other parts of the system start failing. It is possible to put a limitation on much the microservices can scale, which is called throttling. First, the new FPI was measured without throttling it (or putting the throttling at such a high limit, that the other parts of the systems still started failing) to find its extremes. Based on those results the microservices were throttled by half of its extremes and measured with that throttling. Summarized, there are three measurements; Old FPI, New FPI MAX (extremes) and new FPI.

The measurements are divided into three phases; reading data, e-commerce index process, and search index process. The reading data measures the time it takes for the FPI to read all data that is provided by *nightly full dumps*. The e-commerce indexer measures the time it takes to transform and save all the data to the e-commerce database. The search indexer process measures the time it takes to transform and save all the data to the search database.

4.1 Reading Data Measurements

In the old FPI, the reading of the data in *nightly full dumps* is part of the e-commerce indexer, but it is two separated services in the new FPI. The *nightly full dumps* are stored in infinitely scalable data storage, where the reading of them is only limited by the bandwidth speed between services. Throttling this part did not make sense for the new FPI, so only one scenario was measured. As shown in Figure 6, the data reading is roughly five times faster to finish with the new FPI. During the first unit of time, the new FPI only reads roughly 30% of the data because the service takes a few seconds to start up.

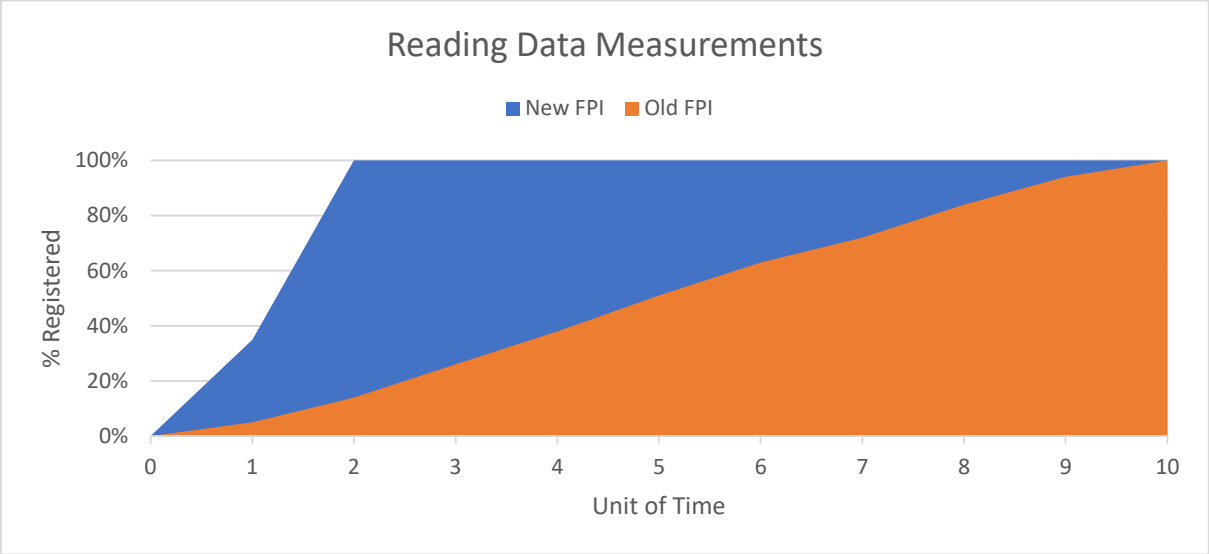

Figure 6 Reading data measurements chart

4.2 E-commerce Indexer Measurements

In the old FPI the performance was mostly bottlenecked by the e-commerce database shared between the product import and e-commerce application. However, the new FPI is only shown in the e-commerce database by a small increase in consumption of its total capacity. Instead, the search database had become the new bottleneck. As shown in Figure 7, the new FPI can run significantly faster, but at its max rate, the search application became unusable. Throttling the system by half, made the new FPI index into e-commerce at the same rate as the old FPI.

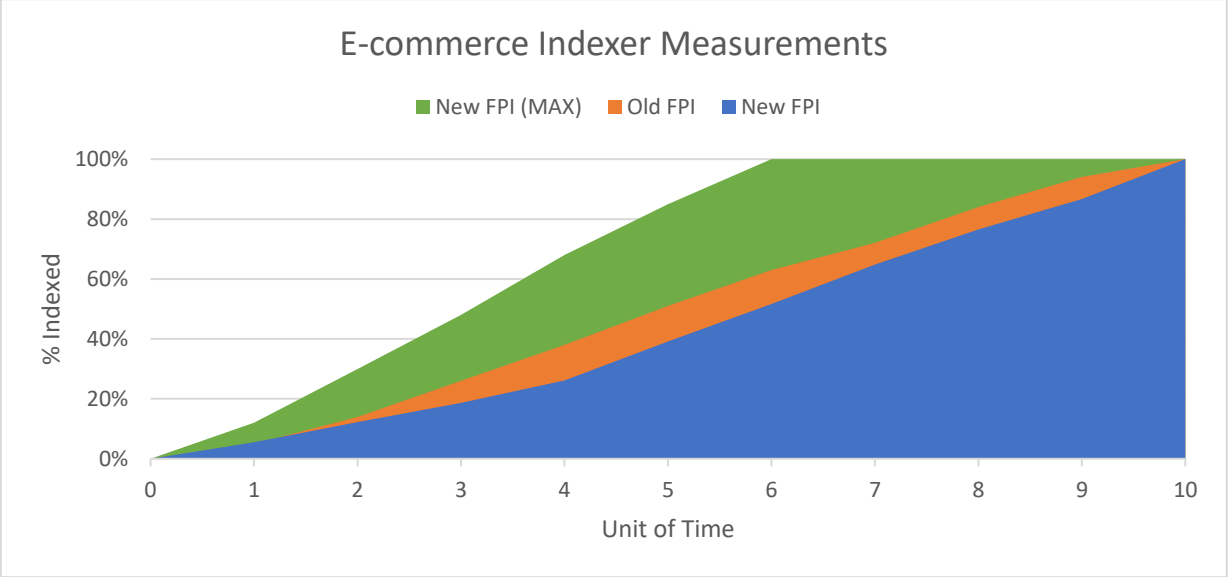

Figure 7 E-commerce indexer measurements chart

4.3 Search Indexer Measurements

As shown in figure 8, the search indexing process is by far the most considerable improvement in the new FPI. The result is partly due to the concurrent indexing of e-commerce and search databases. The highly cohesive microservices also contributed to a more effective search indexing. By removing the search indexer application altogether, a more cost-effective product integration was achieved as well.

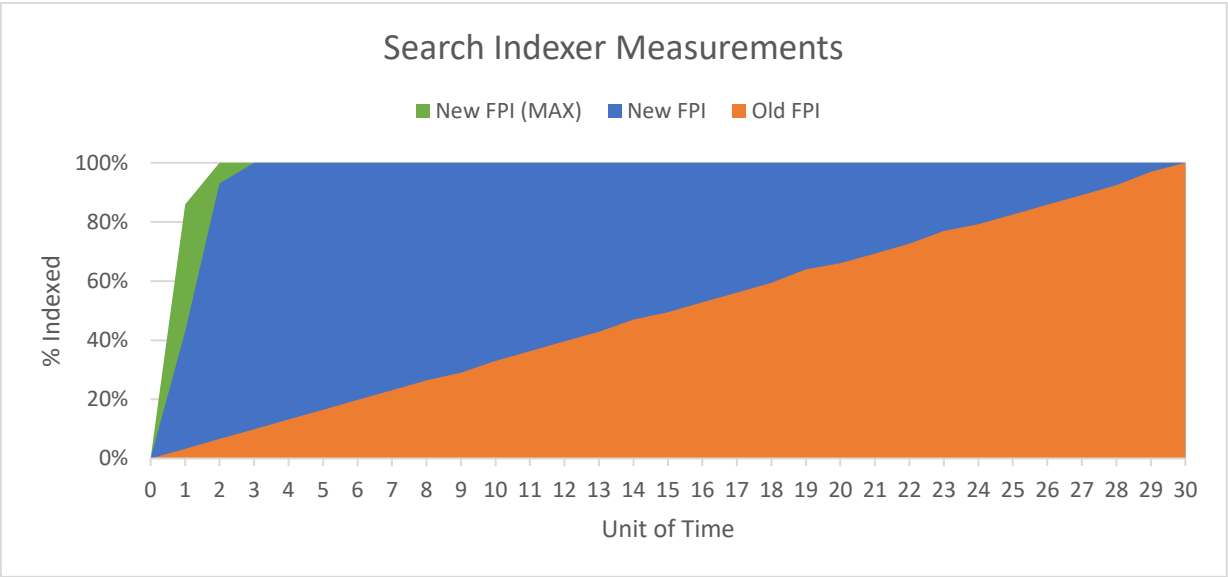

Figure 8 Search indexer measurements chart

5 Discussion

I am delighted and proud of the outcome of the product integration transformation. The new product flow is not entirely decoupled from the rest of the system yet, but it is enough decoupled to enable the product integration to run its development life cycle at its own pace. According to me, the goals were met not just functionally, but also in a highly qualitative manner. The microservices within the new product flow are clearly defined and their sizes are small enough that one would be able to understand them with just a few hours dedicated time.

5.1 Quality Assurance

During the development, I encouraged the team to always dedicate time to write reliable automated tests. With the dedication, the team felt safe introducing changes to the code, because the tests would alert if mistakes were made. In the late stages of development, we introduced drastic changes to improve upon unpredictable problems we had identified during the development. In my experience, drastic changes do not happen in the late stages of the development to avoid introducing many bugs that would delay the deadlines. However, due to the well-written tests, we were able to implement the changes fast and, in most cases, say with confidence that it is bug-free. We did not achieve a full 100% coverage of tested code, but we did have many thorough tests for complex parts of the code, which saved us much time.

Stockmann did a thorough manual end-to-end testing to make sure that the new product flow responds properly to changes in other systems. During the testing, they found a few anomalies in the product data format compared to the old product integration that we had not considered. We were able to fix these fast without introducing further anomalous, which was a good sign of quality. Other than those, the new product flow just worked. Stockmann could already see that their investment in upgrading the product integration was going to pay off.

I learned that naming conventions are more important than you would think in development. It might sound silly at first, but in the long run, you will appreciate it. We chose to adopt a declarative style of programming, where the idea is that the code naming explains what it does. I did not strictly enforce naming conventions from the start, which became code that felt confusing. I brought up the issue of naming and over time we started adapting stricter conventions. The stricter conventions made the code more intuitive and easier to understand.

5.2 Microservices in Reality

I found that one should take the microservice knowledge on the internet with a grain of salt. Microservice architecture has been around for a while and it has many vague and conflicting opinions. The choices of a microservice architecture are delicate and therefore need careful planning before it is approached. I had the fortune to work with talented developers and cloud experts that were able to give valuable feedback on the architectural choices I suggested to approach. In addition, we already had good experience in much of the needed cloud infrastructure, which benefited us a lot already in the design planning. I would not recommend the approach if a team has no prior experience in either service-oriented or microservices architecture, especially not if the budget is limited.

Microservice architecture advocates promote the key benefits of having small services, that can be independently deployed, easy to maintain and isolated from failure. I do not believe any of these problems are magically solved by adopting a microservices architecture. I believe that microservices architecture should be seen as a set of goals that your team can work towards in order to gain the benefits of its style. A team can start with a monolithic application and then slowly start breaking out logical boundaries of it into microservices. Advocates also seem to encourage by starting small, incrementally adding microservices one at a time. I would argue that, yes, microservices tend to be small, but you need to think big when introducing them.

5.3 Future

Both Stockmann and the developers are satisfied with the outcome of the architectural choice. Developers feel it is easy to pick up and comfortable environment to develop in. Stockmann has started considering the architecture for other parts of the e-commerce as well and developers are excited to solve new challenges with microservices. The new product flow has already been in production for several months and is performing great. Stockmann has a bi-annual event called *crazy days* which is a big campaign that usually caused problems to surface in the e-commerce site due to misfunctioning product integration. However, the new product flow was active the latest *crazy days* and the team did not observe a single problem in relation to the product integration. I believe all the above mentioned achieved goals prove that the product integration is now ready for Stockmann’s digital future.

6 Bibliography

Amazon, 2018. *What is Cloud Computing? - Amazon Web Services*. [online] Available at: <https://aws.amazon.com/what-is-cloud-computing/> [Accessed 6 Jul. 2018].

AOE GmbH, 2017. *Case Study : Stockmann Omnichannel Commerce across all Channels The Challenge*. [online] Available at: <https://www.aoe.com/en/clients/stockmann-000458.html> [Accessed 20 Apr. 2019].

Avram, A. and Marinescu, F., 2006. *Domain-Driven Design Quickly*.

Dhiman, R., 2015. *Microservices Architecture | Pluralsight*. [online] Pluralsight. Available at: <https://www.pluralsight.com/courses/microservices-architecture> [Accessed 6 Jul. 2018].

Dictionary.com, 2018. *Monolith | Define Monolith at Dictionary.com*. [online] Available at: <http://www.dictionary.com/browse/monolith> [Accessed 6 Jul. 2018].

Digia Plc, 2018a. *Company Strategy of Digia*. [online] Available at: <https://digia.com/en/company/strategy/> [Accessed 9 Jul. 2018].

Digia Plc, 2018b. *Digia as a Company*. [online] Available at: <https://digia.com/en/company/> [Accessed 9 Jul. 2018].

Erl, T., 2016. *Service-Oriented Architecture: Analysis and Design for Services and Microservices*. 2nd ed. Prentice Hall.

Erl, T., Puttini, R. and Mahmood, Z., 2013. *Cloud Computing: Concepts, Technology & Architecture*. 1st ed. [online] Prentice Hall. Available at: <http://www.informit.com/store/cloud-computing-concepts-technology-architecture-9780133387520>.

Evans, E., 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

Fowler, M., 2004. *IntegrationDatabase*. [online] Available at: <https://www.martinfowler.com/bliki/IntegrationDatabase.html> [Accessed 30 Nov. 2018].

Fowler, M., 2015. *MonolithFirst*. [online] Available at:

<https://martinfowler.com/bliki/MonolithFirst.html> [Accessed 7 Jul. 2018].

Fowler, M. and Lewis, J., 2014. *Microservices - A definition of this new architectural term*. [online] Available at: <https://martinfowler.com/articles/microservices.html> [Accessed 22 Sep. 2018].

Ingeno, J., 2018. *Software Architect Handbook*. Packt Publishing Ltd.

Len, B., Clements, P. and Kazman, R., 2012. *Software Architecture in Practice: Software Architect Practice, Edition 3*. Addison-Wesley.

Morris, B., 2013. *A shared database is still an anti-pattern, no matter what the justification*. [online] Available at: <https://www.ben-morris.com/a-shared-database-is-still-an-anti-pattern-no-matter-what-the-justification/> [Accessed 29 Nov. 2018].

Nadareishvili, I., Mitra, R., McLarty, M. and Amundsen, M., 2016. *Microservice Architecture: Aligning Principles, Practices, and Culture*. *Microservices, IoT, and Azure*.

Neto, M.D., 2014. *A brief history of cloud computing - Cloud computing news*. [online] IBM. Available at: <https://www.ibm.com/blogs/cloud-computing/2014/03/18/a-brief-history-of-cloud-computing-3/> [Accessed 6 Jul. 2018].

Newman, S., 2015. *Building Microservices: Designing Fine-Grained Systems*. [online] O'Reilly Media, Inc. Available at: <http://shop.oreilly.com/product/0636920033158.do>.

Pacheco, V.F., 2018. *Microservice Patterns and Eest Practices*. Packt Publishing Ltd.

Stockmann plc, 2018. *Our Year 2018*. [online] Available at: <http://year2018.stockmanngroup.com/> [Accessed 20 Apr. 2019].

Stockmann plc, 2019. *Stockmann: Company information*. [online] Available at: <http://www.stockmanngroup.com/en/company-information> [Accessed 19 Apr. 2019].

Veijalainen, L., 2018. Interim Report. p.18.

Vernon, V., 2016. *Domain-Driven Design Distilled*. Addison-Wesley Professional.

Watts, S., 2017. *SaaS vs PaaS vs IaaS: What's The Difference and How To Choose – BMC*

Blogs. [online] bmc. Available at: <https://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose/> [Accessed 6 Jul. 2018].